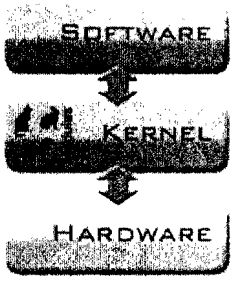
Attachment I
Page 1 of 5

Kernel (computer science)

From Wikipedia, the free encyclopedia
(Redirected from Operating system kernel)

In computing, the **kernel** is the central component of most computer operating systems (OSs). Its responsibilities include managing the system's resources and the communication between hardware and software components. As a basic component of an operating system, a kernel provides the lowest level of abstraction layer for the resources (especially memory, processors and I/O devices) that applications must control to perform their function. It typically makes these facilities available to application processes through inter-process communication mechanisms and system calls.



A kernel connects the software and hardware of a computer

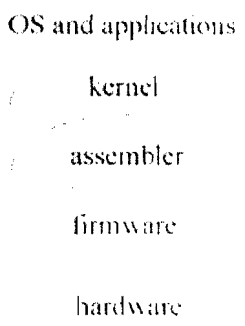
These tasks are done differently by different kernels, depending on their design and implementation. While monolithic kernels will try to achieve these goals by executing all the code in the same address space to increase the performance of the system, microkernels run most of their services in user space, aiming to improve maintainability and modularity of the codebase.[1] A range of possibilities exists between these two extremes.

Contents

Attachment I
page 2 of 5

Overview

Most operating systems rely on the kernel concept. The existence of a kernel is a natural consequence of designing a computer system as a series of abstraction layers[2], each relying on the functions of layers beneath itself. The kernel, from this viewpoint, is simply the name given to the lowest level of abstraction that is implemented in software. In order to avoid having a kernel, one would have to design all the software on the system to not use abstraction layers; this would increase the complexity of the design to such a point that only the simplest systems could feasibly be implemented.



A typical vision of a computer architecture as a series of abstraction layers: hardware, firmware, assembler, **kernel**, operating system and applications (see also Tanenbaum 79).

While it is today mostly called the *kernel*, the same part of the operating system has also in the past been known as the ***nucleus***, ***core***[3][4][5][6] or ***supervisor***. (Note, however, the term *core* has also been used to refer to the primary memory of a computer system, typically because the original memory of computers were a type of magnetic "donut" (a "core") connected at the interection of two wires.)

In most cases, the boot loader starts executing the kernel in supervisor mode,[7] The kernel then initializes itself and starts the first process. After this, the kernel does not typically execute directly, only in response to external events (e.g. via system calls used by applications to request services from the kernel, or via interrupts used by the hardware to notify the kernel of events). Additionally, the kernel typically provides a loop that is executed whenever no processes are available to run; this is often called the *idle process*.

Kernel development is considered one of the most complex and difficult tasks in programming.[8] Its central position in an operating system implies the necessity for good performance, which defines the kernel as a critical piece of software and makes its correct design and implementation difficult. For various reasons, a kernel might not even be able to use the abstraction mechanisms it provides to other software. Such reasons include memory management concerns (for example, a user-mode function might rely on memory being subject to demand paging, but as the kernel itself provides that facility it cannot use it, because then it might not remain in memory to provide that facility) and lack of reentrancy, thus making its development even more difficult for software engineers.

A kernel will usually provide features for low-level scheduling[9] of processes (dispatching), Inter-process communication, process synchronization, context switch, manipulation of process control blocks, interrupt handling, process creation and destruction, process suspension and resumption (see process states)[3][6]

Kernel basic responsibilities

The kernel's primary purpose is to manage the computer's resources and allow other programs to run and use these resources. Typically, the resources consist of:

- The CPU (frequently called the processor). This is the most central part of a computer system, responsible for *running* or *executing* programs on it. The kernel takes responsibility for deciding at any time which of

Attachment E page 3 of 5

the many running programs should be allocated to the processor or processors (each of which can usually run only one program at once)

- The computer's memory. Memory is used to store both program instructions and data. Typically, both need to be present in memory in order for a program to execute. Often multiple programs will want access to memory, frequently demanding more memory than the computer has available. The kernel is responsible for deciding which memory each process can use, and determining what to do when not enough is available.
- Any Input/Output (I/O) devices present in the computer, such as disk drives, printers, displays, etc. The kernel allocates requests from applications to perform I/O to an appropriate device (or subsection of a device, in the case of files on a disk or windows on a display) and provides convenient methods for using the device (typically abstracted to the point where the application does not need to know implementation details of the device)

Kernels also usually provide methods for synchronization and communication between processes (called *inter-process communication* or IPC).

A kernel may implement these features itself, or rely on some of the processes it runs to provide the facilities to other processes, although in this case it must provide some means of IPC to allow processes to access the facilities provided by each other.

Finally, a kernel must provide running programs with a method to make requests to access these facilities.

Process management

The main task of a kernel is to allow the execution of applications and support them with features such as hardware abstractions. To run an application, a kernel typically sets up an address space for the application, loads the file containing the application's code into memory (perhaps via demand paging), sets up a stack for the program and branches to a given location inside the program, thus starting its execution.[10]

Multi-tasking kernels are able to give the user the illusion that the number of processes being run simultaneously on the computer is higher than the maximum number of processes the computer is physically able to run simultaneously. Typically, the number of processes a system may run simultaneously is equal to the number of CPUs installed (however this may not be the case if the processors support simultaneous multithreading).

In a pre-emptive multitasking system, the kernel will give every program a slice of time and switch from process to process so quickly that it will appear to the user as if these processes were being executed simultaneously. The kernel uses scheduling algorithms to determine which process is running next and how much time it will be given. The algorithm chosen may allow for some processes to have higher priority than others. The kernel generally also provides these processes a way to communicate; this is known as inter-process communication (IPC) and the main approaches are shared memory, message passing and remote procedure calls (see concurrent computing).

Other systems (particularly on smaller, less powerful computers) may provide co-operative multitasking, where each process is allowed to run uninterrupted until it makes a special request that tells the kernel it may switch to another process. Such requests are known as "yielding", and typically occur in response to requests for interprocess communication, or for waiting for an event to occur. Older versions of Windows and Mac OS both used co-operative multitasking but switched to pre-emptive schemes as the power of the computers they were targeted to grew.

The operating system might also support multiprocessing (SMP or Non-Uniform Memory Access); in that case, different programs and threads may run on different processors. A kernel for such a system must be designed to be re-entrant, meaning that it may safely run two different parts of its code simultaneously. This typically means providing synchronization mechanisms (such as spinlocks) to ensure that no two processors attempt to modify the

Attachment I
Page 4 of 5

same data at the same time.

Memory management

The kernel has full access to the system's memory and must allow processes to access this memory safely as they require it. Often the first step in doing this is virtual addressing, usually achieved by paging and/or segmentation. Virtual addressing allows the kernel to make a given physical address appear to be another address, the virtual address. Virtual address spaces may be different for different processes; the memory that one process accesses at a particular (virtual) address may be different memory from what another process accesses at the same address. This allows every program to behave as if it is the only one (apart from the kernel) running and thus prevents applications from crashing each other.[10]

On many systems, a program's virtual address may refer to data which is not currently in memory. The layer of indirection provided by virtual addressing allows the operating system to use other data stores, like a hard drive, to store what would otherwise have to remain in main memory (RAM). As a result, operating systems can allow programs to use more memory than the system has physically available. When a program needs data which is not currently in RAM, the CPU signals to the kernel that this has happened, and the kernel responds by writing the contents of an inactive memory block to disk (if necessary) and replacing it with the data requested by the program. The program can then be resumed from the point where it was stopped. This scheme is generally known as demand paging.

Virtual addressing also allows creation of virtual partitions of memory in two disjointed areas, one being reserved for the kernel (kernel space) and the other for the applications (user space). The applications are not permitted by the processor to address kernel memory, thus preventing an application from damaging the running kernel. This fundamental partition of memory space has contributed much to current designs of actual general-purpose kernels and is almost universal in such systems, although some research kernels (e.g. Singularity) take other approaches.

Device management

To perform useful functions, processes need access to the peripherals connected to the computer, which are controlled by the kernel through device drivers. For example, to show the user something on the screen, an application would make a request to the kernel, which would forward the request to its display driver, which is then responsible for actually plotting the character/pixel.[10]

A kernel must maintain a list of available devices. This list may be known in advance (e.g. on an embedded system where the kernel will be rewritten if the available hardware changes), configured by the user (typical on older PCs and on systems that are not designed for personal use) or detected by the operating system at run time (normally called Plug and Play).

In a plug and play system, a device manager first performs a scan on different hardware buses, such as Peripheral Component Interconnect (PCI) or Universal Serial Bus (USB), to detect installed devices, then searches for the appropriate drivers.

As device management is a very OS-specific topic, these drivers are handled differently by each kind of kernel design, but in every case, the kernel has to provide the I/O to allow drivers to physically access their devices through some port or memory location. Very important decisions have to be made when designing the device management system, as in some designs accesses may involve context switches, making the operation very CPU-intensive and easily causing a significant performance overhead.

System calls

Attachment I
Page 5 of 5

To actually perform useful work, a process must be able to access the services provided by the kernel. This is implemented differently by each kernel, but most provide a C library or an API, which in turn invoke the related kernel functions.

The method of invoking the kernel function varies from kernel to kernel. If memory isolation is in use, it is impossible for a user process to call the kernel directly, because that would be a violation of the processor's access control rules. A few possibilities are:

- Using a software-simulated interrupt. This method is available on most hardware, and is therefore very common.
- Using a call gate. A call gate is a special address which the kernel has added to a list stored in kernel memory and which the processor knows the location of. When the processor detects a call to that location, it instead redirects to the target location without causing an access violation. Requires hardware support, but the hardware for it is quite common.
- Using a special system call instruction. This technique requires special hardware support, which common architectures (especially x86) may lack. System call instructions have been added to recent models of x86 processors, however, and some (but not all) operating systems for PCs make use of them when available.
- Using a memory-based queue. An application that makes large numbers of requests but does not need to wait for the result of each may add details of requests to an area of memory that the kernel periodically scans to find requests.

Kernel design decisions

Fault tolerance

An important consideration in the design of a kernel is fault tolerance; specifically, in cases where multiple programs are running on a single computer, it is usually important to prevent a fault in one of the programs from negatively affecting the other. Extended to malicious design rather than a fault, this also applies to security, and is necessary to prevent processes from accessing information without being granted permission.

Two main approaches to the protection of sensitive information are assigning privileges to hierarchical protection domains, for example by using a processor's supervisor mode, or distributing privileges differently for each process and resource, for example by using capabilities or access control lists.

Hierarchical protection domains are much less flexible, as it is not possible to assign different privileges to processes that are at the same privileged level, and can't therefore satisfy Denning's four principles for fault tolerance (particularly the Principle of least privilege). Hierarchical protection domains also have a major performance drawback, since interaction between different levels of protection, when a process has to manipulate a data structure both in 'user mode' and 'supervisor mode', always requires message copying (transmission by value)[11]. A kernel based on capabilities, however, is more flexible in assigning privileges, can satisfy Denning's fault tolerance principles[12], and typically doesn't suffer from the performance issues of copy by value.

Both approaches typically require some hardware or firmware support to be operable and efficient. The hardware support for hierarchical protection domains[13] is typically that of "CPU modes". An efficient and simple way to provide hardware support of capabilities is to delegate the MMU the responsibility of checking access-rights for every memory access, a mechanism called capability-based addressing[12]. Most commercial computer architectures lack MMU support for capabilities. An alternative approach is to simulate capabilities using commonly-support hierarchical domains; in this approach, each protected object must reside in an address space that the application does not have access to; the kernel also maintains a list of capabilities in such memory. When an application needs to access an object protected by a capability, it performs a system call and the kernel

Attachment II
page 1 of 1

Spinlock

From Wikipedia, the free encyclopedia

In software engineering, a **spinlock** is a lock where the thread simply waits in a loop ("spins") repeatedly checking until the lock becomes available. As the thread remains active but isn't performing a useful task, the use of such a lock is a kind of busy waiting. Once acquired, spinlocks will usually be held until they are explicitly released, although in some implementations they may be automatically released if the thread blocks (aka "goes to sleep").

Spinlocks are efficient if threads are only likely to be blocked for a short period of time, as they avoid overhead from operating system process re-scheduling or context switching. For this reason, spinlocks are often used inside operating system kernels. However, spinlocks become wasteful if held for longer, both preventing other threads from running and requiring re-scheduling. Spinlocks are always inefficient on single-processor systems - no other thread is able to make progress while the waiting thread spins.

Implementing spinlocks is difficult, because one must take account of the possibility of simultaneous access to the lock to prevent race conditions. Generally this is only possible with special assembly language instructions, such as atomic test-and-set operations, and cannot be implemented from high level languages like C.[1] On architectures without such operations, or if high-level language implementation is required, a non-atomic locking algorithm may be used, e.g. Peterson's algorithm. But note that such an implementation may require more memory than a spinlock, be slower to allow progress after unlocking, and may not be implementable in a high-level language if out-of-order execution is in use.

Contents

Example implementation

The following example uses x86 assembly language to implement a spinlock. It will work on any Intel 80386 compatible processor.

```
lock:                   # The lock variable. 1 = locked, 0 = unlocked.
    dd      0

spin_lock:
    mov     eax, 1      # Set the EAX register to 1.

loop:
    xchg    eax, [lock] # Atomically swap the EAX register with
                        #  the lock variable.
                        # This will always store 1 to the lock, leaving
                        #  previous value in the EAX register.

    test    eax, eax    # Test EAX with itself. Among other things, this will
                        #  set the processor's Zero Flag if EAX is 0.
                        # If EAX is 0, then the lock was unlocked and
                        #  we just locked it.
```